PRIFYSGOL ABERYSTWYTH UNIVERSITY

CS10720 Problems and Solutions

Thomas Jansen

Today: Sorting: Heapsort

February 22nd

Plans for Today

1. Problem Definition
   - Motivation
2. Heaps
   - Using an Array as a Data Structure
   - Max Heaps
3. Heapsort
   - Idea and Framework
   - Implementation
4. Analysing Heapsort
   - Analysis
   - Application
5. Summary
   - Summary & Take Home Message

Our Problem for Today

Remember we can sort an array using Insertion Sort
with only $O(n \log n)$ key comparisons
but with $\Theta(n^2)$ operations
in the worst case

Remember we will see from the practicals
in average case (input is a random array) quite fast
but in worst case (input is 'wrongly sorted') really slow

Can we find a sorting algorithm that is always fast?

Facts (without proofs (unless you really insist))

- $\Theta(n \log n)$ key comparisons is the best you can achieve (in the worst case and the average case) if you sort using key comparisons
- $\Theta(n \log n)$ operations in total achievable (in the worst case)
- not completely trivial

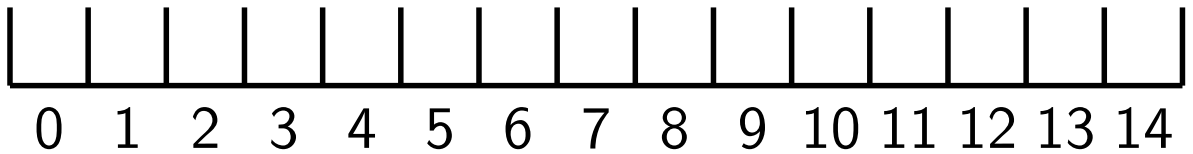
Using an Array as Something Else

Towards a clever data structure...

Remember arrays: `long A[n]; /* n is size of the array */`

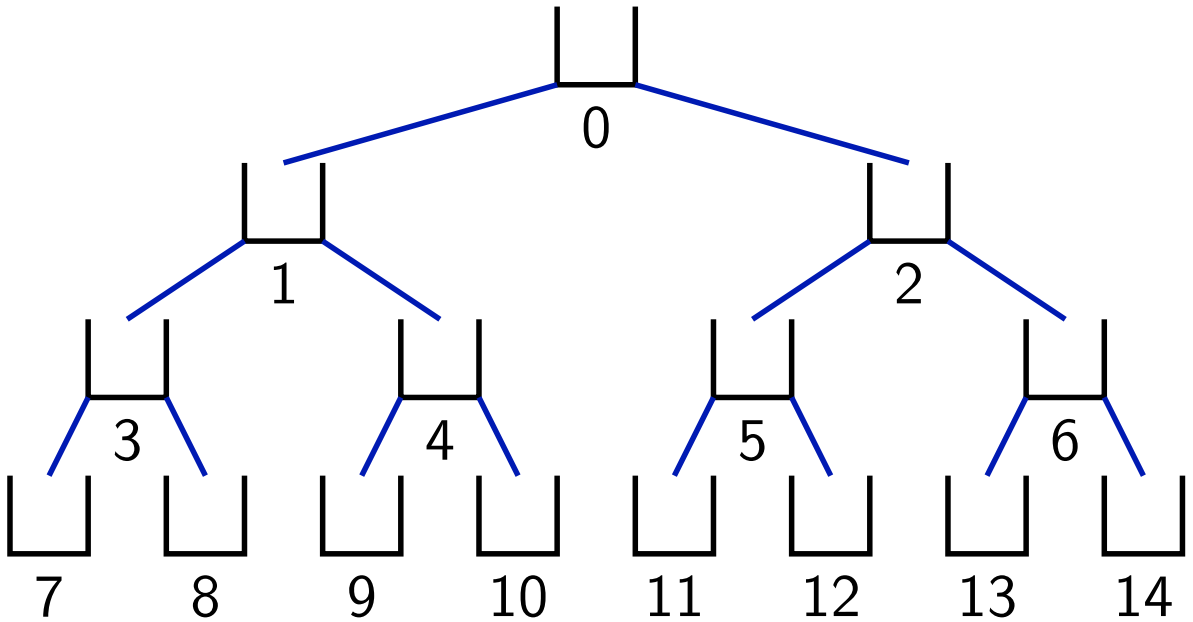
Example

for $n = 15$:



Example

for $n = 15$:



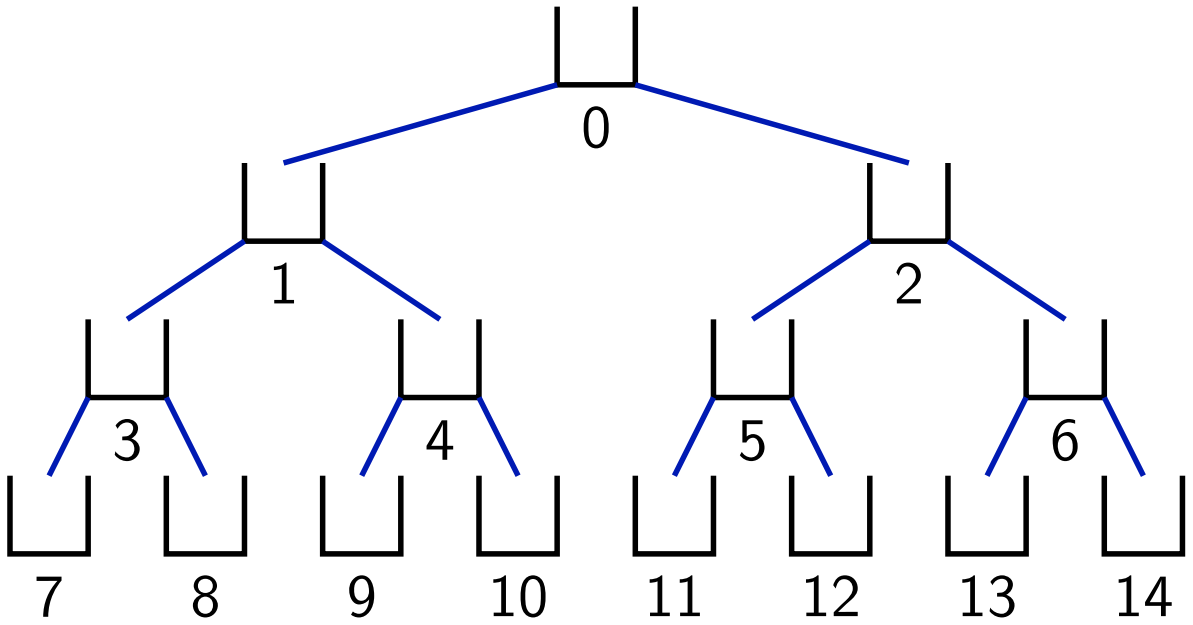
parent	$i \to 2i + 1$	left child
parent	$i \to 2i + 2$	right child
child	$j \to \lfloor (j-1)/2 \rfloor$	parent

A Binary Tree in an Array

Remember

Example

for $n = 15$:



Fact for each node i we can compute

- its parent as $\lfloor (i-1)/2 \rfloor$
- its left child as $2i+1$
- its right child as $2i+2$

Max Heaps

Definition

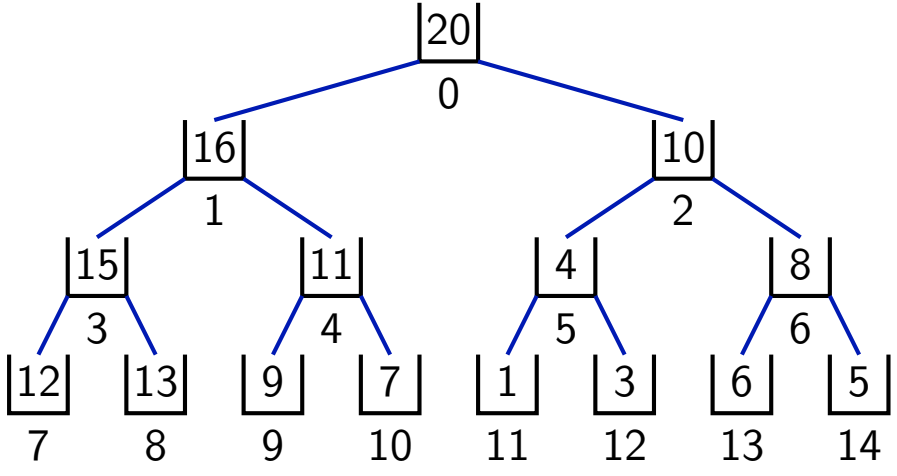
A max heap is a binary tree where the contents of each node is as at least as large as the contents of both of its children.

For the sake of completeness (you know what this is)

Definition

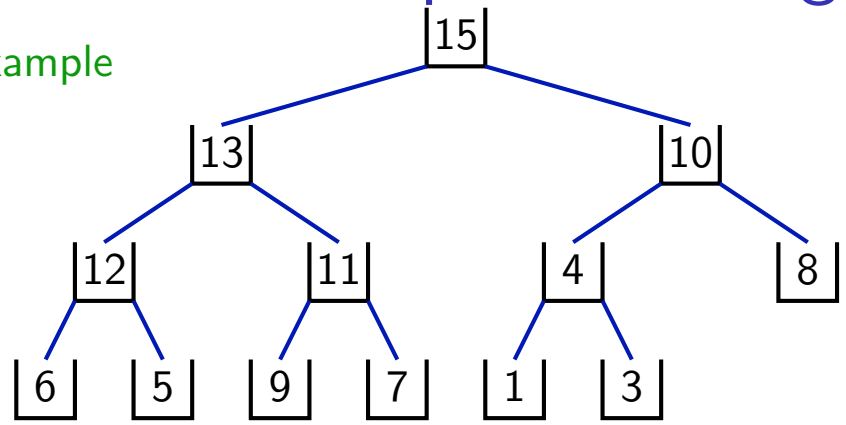
The empty tree ε is a binary tree. If T_1 and T_2 are binary trees then (v, T_1, T_2) is a binary tree with the single node v as root, the root of T_1 as its left child and the root of T_2 as its right child.

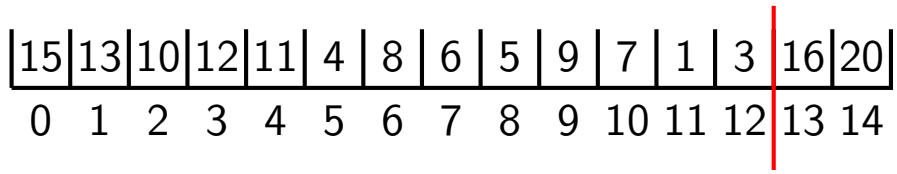
Example



From Max Heaps to Sorting

Example





Observation maximum of array is at root of the max heap
its correct position is at the very end of the array

Idea let's swap it to this position
and pretend the array is one element shorter

Observation max heap property violated

Idea find good candidates for repair
by following the path of larger children to the bottom
and climbing it back up, making swaps where needed

Repeat!

Implementing Heapsort

```
void heapSort(long *keys, long size) {
  long i; /* index variable to run over array */
  long currentSize; /* keep track of remaining heap size */
  long swap; /* memory to keep element for swap */
  /* build heap */
  Missing for the time being.
  /* sort */
  for ( currentSize = size; currentSize>0; currentSize-- ) {
    /* swap root with last element */
    swap = keys[0];
    keys[0] = keys[currentSize-1];
    keys[currentSize-1] = swap;
    /* repair heap */
    reheap(keys, 0, currentSize-1);
  }
}
```

Towards Implementing `reheap`

```
void reheap(long *keys, long root, long size) {
  /* compute path of larger children */
  /* climb path back up and swap where needed */
}
```

Remember children of i at $2i+1$ and $2i+2$
no child if $2i+1 \geq$ size
only one child if $2i+2 =$ size

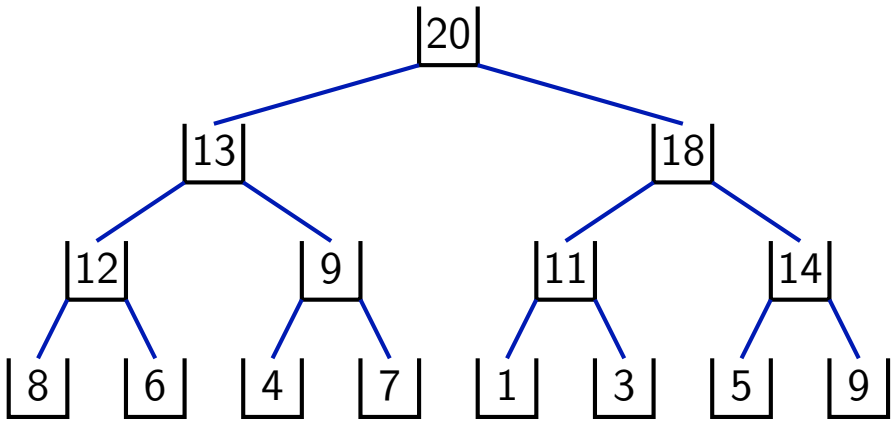
Remember parent of j at $\lfloor (j-1)/2 \rfloor$

Implementing reheap

```
void reheap(long *keys, long root, long size) {
  long i; /* index variable to run over array */
  long swap; /* memory to keep element for swap */
  long key; /* current key that may need new position */
  /* compute path of larger children */
  i=root; /* start at root */
  while ( (2*i+1)<size ) { /* does left child exist?  */
    /* go to larger child */
    if ( (2*i+2<size) && (keys[2*i+1] < keys[2*i+2]) )
      i = 2*i+2; /* move to right child */
    else
      i = 2*i+1; /* move to left child */
  }
  /* climb path back up and find where swaps are needed */
  while ( (i>root)&&(keys[root]>=keys[i])) {
    /* look for key > root */
    i=(i-1)/2; /* move path up */
  }
  /* perform swaps */
  key = keys[root]; /* remember root */
  while ( i>root) ) {
    swap=keys[i];
    keys[i]=key;
    key=swap;
    i=(i-1)/2; /* move path up */
  }
  keys[root] = key; /* put key in root */
}
```

Building a Max Heap

Example



8	20	18	13	9	11	14	12	6	4	7	1	3	5	9
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Observation We can read any node as root of a max heap starting at that node and ending at the end of the array

Observation to build up the max heap it is sufficient to call `reheap(keys, i, size)` for every node $i \in \{0, 1, \ldots, \lfloor \texttt{size}/2 \rfloor\}$ in reverse order

Implementing Heapsort

```
void heapSort(long *keys, long size) {
  long i; /* index variable to run over array */
  long currentSize; /* keep track of remaining heap size */
  long swap; /* memory to keep element for swap */
  /* build heap */
  for ( i=size/2; i>=0; i-- ) {
    reheap(keys, i, size); /* repair heap[i..size-1] */
  }
  /* sort */
  for ( currentSize = size; currentSize>0; currentSize-- ) {
    /* swap root with last element */
    swap = keys[0];
    keys[0] = keys[currentSize-1];
    keys[currentSize-1] = swap;
    /* repair heap */
    reheap(keys, 0, currentSize-1);
  }
}
```

Analysing Heapsort: Part 1: reheap

```
void reheap(long *keys, long root, long size) {
  long i; /* index variable to run over array */
  long swap; /* memory to keep element for swap */
  long key; /* current key that may need new position */
  /* compute path of larger children */
  i=root; /* start at root */
  while ( (2*i+1)<size ) { /* does left child exist? */
    /* go to larger child */
    if ( (2*i+2<size) && (keys[2*i+1] < keys[2*i+2]) )
      i = 2*i+2; /* move to right child */
    else
      i = 2*i+1; /* move to left child */
  }
  /* climb path back up and find where swaps are needed */
  while ( (i>root)&&(keys[root]>=keys[i])) {
    /* look for key > root */
    i=i/2; /* move path up */
  }
  /* perform swaps */
  key = keys[root]; /* remember root */
  while ( i>root) ) {
    swap=keys[i];
    keys[i]=key;
    key=swap;
    i=i/2; /* move path up */
  }
  keys[root] = key; /* put key in root */
}
```

Worst case time:
Θ(height of heap)

Worst case time:
Θ(height of heap)

Analysing Heapsort: Part 2: Sorting Phase

```
void heapSort(long *keys, long size) {
  long i; /* index variable to run over array */
  long currentSize; /* keep track of remaining heap size */
  long swap; /* memory to keep element for swap */
  /* build heap */
  for ( i=size/2; i>=0; i-- ) {
    reheap(keys, i, size); /* repair heap[i..size-1] */
  }
  /* sort */
  for ( currentSize = size; currentSize>0; currentSize-- ) {
    /* swap root with last element */
    swap = keys[0];
    keys[0] = keys[currentSize-1];
    keys[currentSize-1] = swap;
    /* repair heap */
    reheap(keys, 0, currentSize-1);
  }
}
```

Observation worst case time dominated by loop body of loop dominated by `reheap`

Observation height of heap always $O(\log n)$

Consequence worst case time $O(n \log n)$

Analysing Heapsort: Part 3: Heap Creation Phase

```
void heapSort(long *keys, long size) {
  long i; /* index variable to run over array */
  long currentSize; /* keep track of remaining heap size */
  long swap; /* memory to keep element for swap */
  /* build heap */
  for ( i=size/2; i>=0; i-- ) {
    reheap(keys, i, size); /* repair heap[i..size-1] */
  }
  /* sort */
  for ( currentSize = size; currentSize>0; currentSize-- ) {
    /* swap root with last element */
    swap = keys[0];
    keys[0] = keys[currentSize-1];
    keys[currentSize-1] = swap;
    /* repair heap */
    reheap(keys, 0, currentSize-1);
  }
}
```

Observation worst case time determined by loop

$$O\left(\frac{n}{4}\cdot 1+\frac{n}{8}\cdot 2+\frac{n}{16}\cdot 3+\cdots+1\cdot \text{depth}\right)$$
$$=O\left(\frac{n}{2^2}\cdot 1+\frac{n}{2^3}\cdot 2+\frac{n}{2^4}\cdot 3+\cdots+\frac{n}{2^{\log_n 2}}\cdot(\log_2(n)+1)\right)$$
$$=O\left(n\cdot\left(\frac{1}{2^2}+\frac{2}{2^3}+\frac{3}{2^4}+\cdots+\frac{\log_2(n)+1}{2^{\log_n 2}}\right)\right)=O(n)$$

Analysing Heapsort: Summary

Remember

- worst case run time heap creation time $O(n)$
- worst case run time sorting phase $O(n \log n)$

Consequence worst case run time is $O(n) + O(n \log n) = O(n \log n)$
and number of all operations is of same order
as number of key comparisons

Consequence in the worst case,
Heapsort is very much faster than Insertion Sort

What is a worst case instance for Heapsort? Completely unclear.

Coming Back to the Original Motivation

Problem have unsorted array of size n and s searches to perform

Potential solutions

- Perform s linear searches.
  worst case time $\Theta(s \cdot n)$
- (1) Sort the array. (2) Perform s binary searches.
  worst case time $\Theta(\text{Time for sorting} + s \cdot \log n)$
  $= \Theta(n \log(n) + s \cdot \log n)$

Observation approach using search better
if $s = \omega(n \log(n)/(n - \log n))$
(and not worse if $s = \Omega(n \log(n)/(n - \log n))$)
almost certainly useful

size n	**'required searches' $n \log_2(n)/(n - \log_n)$**
10	≈ 5
100	≈ 7
1000	≈ 10
10 000	≈ 13

Summary & Take Home Message

Things to remember

- max heap
- max heap in an array
- Heapsort
- repairing a heap at the root: reheap
- heap creation phase of Heapsort
- sorting phase of Heapsort
- worst case analysis of Heapsort

Take Home Message

- Sorting in time $O(n \log n)$ is possible and worth the effort.
- Heapsort is much more efficient than Insertion Sort for most inputs.

Lecture feedback `http://onlineted.com`